Error Handling and Exceptions

Gabriele Keller
Tom Smeding

Exceptions

- Error handling so far:
  - The `Error` expression to handle run-time errors deterministically aborts the whole program
  - For many applications, this is not the appropriate behaviour
  - Exceptions permit a more fine grained response to run-time errors

- Error:
  - result of a programming error (e.g., preconditions of a function not met), can be fixed by fixing the program

- Exception:
  - result of expected, but irregular occurrence, can possibly be handled in the program

Exceptions

- Exceptions in MinHs:
  - (1) raising (or throwing) an exception: `raise` e
    - e : information about handling of exception
  - (2) catching an exception: `try` e_1 `handle` x `=>` e_2
    - catch expression raised in e_1
    - exception handler is e_2
    - access to information about handling exception via x

Exceptions

- Abstract Syntax
  - `raise e` (Raise e)
  - `try` e_1 `handle` x `=>` e_2 (Try e_1 ($x.e_2$))
- Informal evaluation rules: on `try` e_1 `handle` x `=>` e_2
  - evaluate e_1, and
  - if (`raise` e) is encountered during e_1, evaluate e to value v, bind x to v and then evaluate e_2

Exceptions

- Example:

```
try
  if (y <= 0)
    then raise -1
    else x/y
 handle err =>
    if err = -1
      then 0
   ....
```

- `try` expressions can be nested
  - innermost try expression catches
  - handler may re-raise an exception

Exceptions

- Observations:
  - type of exception values (second argument of raise)
    - in many programming languages, this is a fixed type $\boldsymbol{\tau_{exc}}$
    - may simply be a string or integer (exception code)
    - e.g., subclass `Throwable` in Java

Exceptions - Static Semantics

- Typing rules

$$\frac{\Gamma \vdash e : \tau_{exc}}{\Gamma \vdash (\texttt{Raise}\ e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \cup \{x : \tau_{exc}\} \vdash e_2 : \tau}{\Gamma \vdash (\texttt{Try}\ e_1\ (x.e_2)) : \tau}$$

Exceptions - Dynamic Semantics

- We introduce a new machines state

$$s \ll (\texttt{Raise}\ v)$$

  the machine raises an exception with the exception value v

- First approach:

  on $s \ll (\texttt{Raise}\ v)$

  - propagate exception upwards in the control stack s
  - use first handler encountered

Exceptions - Dynamic Semantics

- Entering a `try` block

$$s \succ (\texttt{Try}\ e_1\ (x.\,e_2)) \ \mapsto_C\ (\texttt{Try}\ \Box\ (x.\,e_2)\ \triangleright\ s) \succ\ e_1$$

- Returning to a `try` block

$$(\texttt{Try}\ \Box\,(x.\,e_2)\ \triangleright\ s \succ v_1 \quad \mapsto_C \quad s \prec v_1$$

- Evaluating a `raise` expression

$$s \succ (\texttt{Raise}\ e) \quad \mapsto_C\ (\texttt{Raise}\ \Box)\ \triangleright\ s \quad \succ\ e$$

- Raising an exception

$$(\texttt{Raise}\ \Box)\ \triangleright\ s \quad \succ\ v \quad \mapsto_C \quad s\ \ll\ (\texttt{Raise}\ v)$$

- Catching an exception

$$(\texttt{Try}\ \Box\ \ (x.e_2))\triangleright\ s \quad \ll\ (\texttt{Raise}\ v) \ \mapsto_C \quad s\ \succ\ e_2\ [x{:=}v]$$

- Propagating an exception

$$f\ \triangleright\ s\ \ll\ (\texttt{Raise}\ v) \ \mapsto_C \quad s\ \ll\ (\texttt{Raise}\ v)$$

Exceptions - Dynamic Semantics

- What is the problem here?
  - efficiency: the frames are popped one by one when an exception is raised
- Second approach
  - how can we jump directly to the appropriate handler?
  - we use an extra handler stack h
  - a handler frame contains
    - a copy of the control stack
    - the handler expression

Exceptions - Dynamic Semantics

- Entering a try block

$$(h,\ k) > (\mathtt{Try}\ e_1\ (x.e_2)) \mapsto_C (\mathtt{Handle}\ k\ (x.e_2) \triangleright h, (\mathtt{Try}\ \square) \triangleright k) \succ e_1$$

- Returning to a try block

$$(\mathtt{Handle}\ k\ (x.e_2) \triangleright h, (\mathtt{Try}\ \square) \triangleright k) \prec v_1 \mapsto_C (h,\ k) \prec v_1$$

- Evaluating a raise expression

$$(h,\ k) \succ (\mathtt{Raise}\ e) \mapsto_C (h,\ (\mathtt{Raise}\ \square) \triangleright k) \succ e$$

- Raising an exception

$$(h,\ (\mathtt{Raise}\ \square) \triangleright k) \succ v \mapsto_C (h,\ k) \ll (\mathtt{Raise}\ v)$$

- Catching an exception

$$(\mathtt{Handle}\ k'\ (x.e_2)) \triangleright h,\ k) \ll (\mathtt{Raise}\ v) \mapsto_C (h,\ k') \succ e_2\ [x:=v]$$

How is error/exception handling supported in modern programming languages?

Exception handling in C

- C doesn't have any built in support for exception handling
- Common strategies to handle exceptions:
    - return error value

```c
int divide(int a, int b, int *result) {
    if (b == 0) {
        return -1; // indicate division by zero error
    }
    *result = a / b;
    return 0; // success
}
```

Exception handling in C

- C doesn't have any built in support for exception handling
- Common strategies to handle exceptions:
  - return error value
  - global error code

```c
#include <stdio.h>
#include <errno.h>

int main() {
    FILE *file = fopen("nonexistent.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        printf("Error code: %d\n", errno);
    }
    return 0;
}
```

Exception handling in C

- C doesn't have any built in support for exception handling
- Common strategies to handle exceptions:
  - return error value
  - global error code
  - no language mechanism for stack unwinding in C, but the current call stack can be saved and restored manually (similar to handler stack)

```c
#include <setjmp.h>

jmp_buf env;

void foo() {
    longjmp(env, 1); // jump back to the setjmp point
}

int main() {
    if (setjmp(env) == 0) {
        printf("In the try block\n");
        foo();
    } else {
        printf("Caught an exception\n");
    }
    return 0;
}
```

Exception handling in C

- Using `goto`'s to implement a stack of clean-up actions (e.g., systems code, linux kernel):

```
{

  < do something 1>
  if <error condition 1 >
    goto cleanup1;

  <do something 2>
 if <error condition 2 >
    goto cleanup2;

      ⋮

 cleanup3:
  <undo something 3>;

 cleanup2:
  <undo something 2>;

 cleanup1:
  <undo something 1>;

}
```

Exception handling in C#

- `try` block encapsulates code which may raise an exception
- the `catch` blocks match on exception type:

```
try
{
    // Code that may raise an exception
   ...
   throw new Exception("My custom exception");
   ...
}
catch (DivideByZeroException ex)
{
    // Handle DivideByZeroException
}
catch (Exception ex)
{
    // Handle other exceptions
}
finally {
    // optional cleanup code, executed whether or
    // not exception was raised
}
```

- no matching catch block is encountered after an exception is raised, it will propagate through the call stack, terminating the execution

Exception handling in C#

- The Exception class provides properties, such as `Message`, `StackTrace` that contain information about the exception:

```
catch (Exception ex)
{
    Console.WriteLine($"Exception Message: {ex.Message}");
    Console.WriteLine($"Stack Trace: {ex.StackTrace}");
    // Handle the exception
}
```

Exception handling in Java

- Similar to C# (class `Throwable` corresponds to C# `Exception` class)

```
try {
    // code that may throw an exception or error
} catch (Throwable t) {
    System.out.println("Exception message: " + t.getMessage());
    System.out.println("Stack trace: ");
    t.printStackTrace();
}
```

- Main difference:
  - Java distinguishes between checked and unchecked exceptions
  - if code can throw a checked exception, it must be handled with a `catch`, or declared

```
public void myMethod() throws CheckedException {
    // code that may throw a checked exception
}
```

Error handling in Swift

- Swift uses the type system for safety & clarity:
  - The `throws` keyword in function signatures indicates that the function can throw errors.
  - This makes error-prone areas visible at the call site.
  - A calling function must either handle the error with try or propagate it with `throws`, enforcing clear error boundaries.
  - Different versions of `try` available

Utrecht University

Exception handling in modern languages

- Almost all modern programming languages have built-in support for exception handling
- Java, C#, Swift, Haskell
- Go is somewhat of an exception
  - uses multiple return values, error values
  - no support for call stack unwinding
- Good error and exception handling is an important factor in software quality
  - safety, security and robustness of code
  - helps with debugging and maintenance
  - user experience

Error and Exception handling

- Good error and exception handling are essential, not just for reliability, but also for security!
- Why?
  - many high-profile exploits are due to poor error handling and unchecked undefined behaviour

Exploits due to poor error handling

- Heartbleed (OpenSSL, 2014)
  - improper error handling exacerbated its impact.
  - function processing heartbeat requests did not handle malformed inputs gracefully.
- Exploit:
  - attackers could send malformed heartbeat messages to extract sensitive data from server memory (private keys, user credentials)

Exploits due to poor error handling

- Cloudbleed (2017)
  - Cloudflare used a custom HTML parser written in C, which relied on a buffer to process data
  - buffer overflow under certain conditions
  - memory outside the buffer was exposed in HTTP responses
  - the bug was triggered by specific sequences of malformed HTML or improperly formatted data
  - sensitive data leaked due to undefined behaviour of C

Exploits due to poor error handling

- "goto fail;" vulnerability in Apple's SSL/TLS implementation(2014)

check the cryptographic signature of the server's SSL/TLS certificate

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

fail:
  ...
  return (err);
```

validate that the certificate was properly signed by a trusted Certificate Authority

Exploits due to poor error handling

- Good error messages are important
  - but be aware of information leak!
- Microsoft's ASP.NET Padding Oracle Attack (2010)
  - ASP.NET's exception messages provided detailed error information about cryptographic padding
  - attackers could use this information to perform a padding oracle attack, decrypting encrypted data without the encryption key.
  - allowed attackers to compromise encrypted cookies and view sensitive application data.

Error and exception handling

- Common faults:
  - uncaught exceptions: allowing exceptions to propagate unchecked can leave systems vulnerable.
  - no input validation: many vulnerabilities result from bad handling of malformed or malicious input.
  - (too) detailed error messages: too much information in exceptions can guide attackers.
  - silent failures: failing to act on critical errors can bypass security controls.

Error and exception handling

- Programming languages can't solve the problem, but they can make it
  - impossible to have undefined behaviour
  - more convenient to handle errors
  - harder to miss handling an error
  - easier to spot unhandled errors
- For unsafe languages, tools can be used to check program properties